

Four candidates for Exascale HEC I/O & Lessons from a Persistent Memory System

Raju Rangaswami

School of Computing and Information Sciences
College of Engineering and Computing
Florida International University



HECIWG 2011 Workshop
August 9th, 2011

4 Candidates
●○○○

Context
○

Exploration
○○○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○○

1. Hierarchies

1. Hierarchies

... Are Coming!

1. Hierarchies

... Are Coming!

- ▶ Flash (PCIe, SAS/FC, MLC/SLC)
- ▶ Newer flash is also competitive for sequential I/O
- ▶ Flash attractive at both at compute and storage ends
- ▶ SLC (more erasures) at compute and MLC (cheaper) at storage
- ▶ There may be others ... (but do not meet the Fry's test yet!)
- ▶ We will need disks for capacity (**death of disk?**)
- ▶ Technology-agnostic dynamic management is going to critical

4 Candidates
○●○○

Context
○

Exploration
○○○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○○

2. Asynchrony

2. Asynchrony

... and non-blocking I/O

2. Asynchrony

... and non-blocking I/O

- ▶ Invisible elephant in the room (no one brings it up anymore)
- ▶ But, all we have learned about it is good
- ▶ Allows non-blocking executions and computation to overlap with I/O
- ▶ Systematically explore this single-node performance
 - ✓ I. Asynchronous checkpointing
 - ✓ II. Take it further – “barrierless checkpointing”
- ▶ Break-up slower tasks dynamically ... or
- ▶ ... use on-the-fly redundant computations to tolerate hardware noise/faults
- ▶ How does this affect jitter (?) (Measurement panel studies)

4 Candidates
○○●○

Context
○

Exploration
○○○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○○

3. Out-of-core Execution

3. Out-of-core Execution

Scaling-up single node performance

3. Out-of-core Execution

Scaling-up single node performance

- ▶ ahem ... Virtual memory?
- ▶ Changing hierarchy with flash makes this more attractive
- ▶ Addresses memory pressure
- ▶ Local flash can contain large memory jobs less expensively
- ▶ 100K random IOPS *4KB/sec → 0.5 GBps sustained residual bandwidth for swap.
- ▶ And you can have multiple PCIe/SSD flash devices
- ▶ Improving in every successive generation of flash
- ▶ And getting cheaper too!
- ▶ How about jitter?

4 Candidates
○○○●

Context
○

Exploration
○○○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○○

4. Persistent Memory

4. Persistent Memory

Deja-vu?

4. Persistent Memory

Deja-vu?

- ▶ Applications are embedding I/O codes
- ▶ This cannot be good —
 - ✓ code complexity
 - ✓ optimality
 - ✓ portability
- ▶ Application managed I/O likely worsening with scale
- ▶ What about memory-like interfaces and moving I/O to middleware
- ▶ More on this next ...

Context

An HEC application's storage view

- ▶ Virtualized files (that hide a complex storage system)
- ▶ Optimizing I/O (for checkpointing or otherwise) is non-trivial
- ▶ Applications serialize data and interact with storage
- ▶ Developers track persistent data manually
- ▶ Increased code complexity; reduced reliability and portability

Context

An HEC application's storage view

- ▶ Virtualized files (that hide a complex storage system)
- ▶ Optimizing I/O (for checkpointing or otherwise) is non-trivial
- ▶ Applications serialize data and interact with storage
- ▶ Developers track persistent data manually
- ▶ Increased code complexity; reduced reliability and portability

Taking a step back

Context

An HEC application's storage view

- ▶ Virtualized files (that hide a complex storage system)
- ▶ Optimizing I/O (for checkpointing or otherwise) is non-trivial
- ▶ Applications serialize data and interact with storage
- ▶ Developers track persistent data manually
- ▶ Increased code complexity; reduced reliability and portability

Taking a step back

- ▶ How about ... **persistent memory** ?

Context

An HEC application's storage view

- ▶ Virtualized files (that hide a complex storage system)
- ▶ Optimizing I/O (for checkpointing or otherwise) is non-trivial
- ▶ Applications serialize data and interact with storage
- ▶ Developers track persistent data manually
- ▶ Increased code complexity; reduced reliability and portability

Taking a step back

- ▶ How about ... **persistent memory** ?
- ▶ Natural programming interface (e.g., seamless chkpointing)
- ▶ Developer does not deal with I/O at all!
- ▶ More than a decade of research (80s and early 90s)

Exploration

So, why are HPC applications not using a persistent memory interface today?

Exploration

So, why are HPC applications not using a persistent memory interface today?

Persistent memory solutions

- ▶ Forerunner: **Recoverable Virtual Memory (RVM)**

Exploration

So, why are HPC applications not using a persistent memory interface today?

Persistent memory solutions

- ▶ Forerunner: **Recoverable Virtual Memory (RVM)**
- ▶ No serialization!
- ▶ No I/O!
- ▶ Getting rid of metadata (?)

Exploration

So, why are HPC applications not using a persistent memory interface today?

Persistent memory solutions

- ▶ Forerunner: **Recoverable Virtual Memory (RVM)**
- ▶ No serialization!
- ▶ No I/O!
- ▶ Getting rid of metadata (?)
- ▶ Developers must track all persistent data
- ▶ Developers must track persistent memory modifications
- ▶ Statically mapped segments (security/portability gap)

4 Candidates
○○○○

Context
○

Exploration
○●○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○○

Software Persistent Memory (*SoftPM*)

Applications allocate, and interact with, persistent memory in much the same way as they work with volatile memory

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ **Containers partition persistent data**
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ **Automatic container discovery**
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ **Atomic and incremental persistence of containers**
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ **Virtualized (portable) storage management**
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ **New worlds: persistent memory versioning and branching**
- ▶ More new worlds: out-of-core (FSes and DBs)

Software Persistent Memory (*SoftPM*)

Central Ideas

- ▶ Redesign the interface for (almost) zero developer effort
- ▶ Persistent data can co-exist (intermingle) with volatile
- ▶ Containers partition persistent data
- ▶ Container roots defined by developer
- ▶ Automatic container discovery
- ▶ Atomic and incremental persistence of containers
- ▶ Type-agnostic persistence for weakly typed languages
- ▶ Virtualized (portable) storage management
- ▶ New worlds: persistent memory versioning and branching
- ▶ More new worlds: out-of-core (FSes and DBs)

Using SoftPM : An Example

```
typedef struct l {
    int v;
    struct l *n;
} list;
► a() {
    list *l = malloc(...);
    l->v = X;
    list *e1 = malloc(...);
    e1->v = Y;
    l->n = e1;
    list *e2 = malloc(...);
    e2->v = Z;
    e1->n = e2;
    :
}
```

Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    ► list *l = malloc(...);  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
►    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
    l->v = X;  
    ► list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
    l->v = X;  
    list *e1 = malloc(...);  
►     e1->v = Y;  
    e1->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Using SoftPM : An Example

```
typedef struct l {
    int v;
    struct l *n;
} list;
a() {
    list *l = malloc(...);
    l->v = X;
    list *e1 = malloc(...);
    e1->v = Y;
    l->n = e1;
    list *e2 = malloc(...);
    e2->v = Z;
    e1->n = e2;
    :
}
```



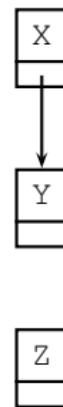
Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



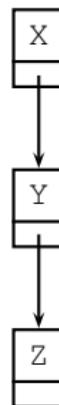
Using SoftPM : An Example

```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
►     e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Using SoftPM : An Example

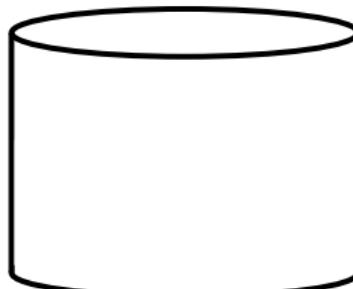
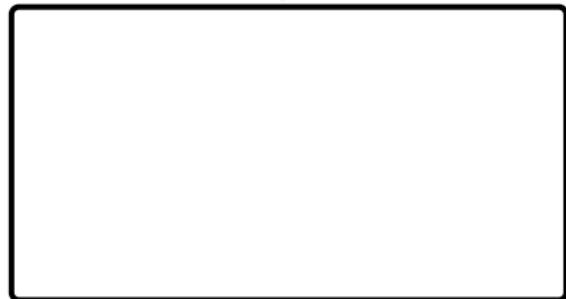
```
typedef struct l {  
    int v;  
    struct l *n;  
} list;  
a() {  
    list *l = malloc(...);  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    :  
}
```



Creating a Persistence Point

```
► struct pcont {  
    list *L;  
} C;  
  
► a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
  
:
```

Process Memory



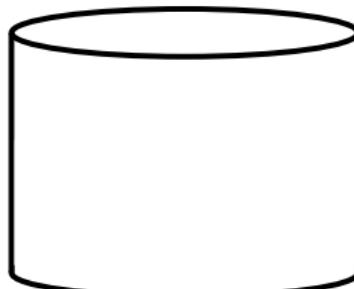
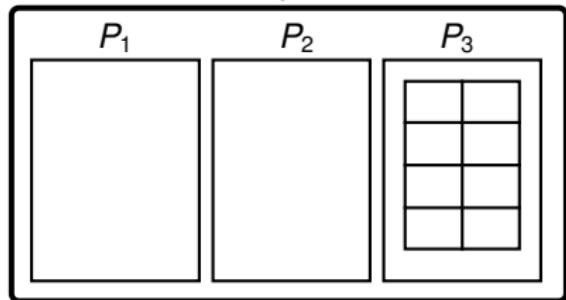
Persistent Medium

Creating a Persistence Point



```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

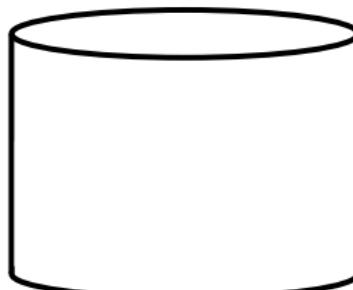
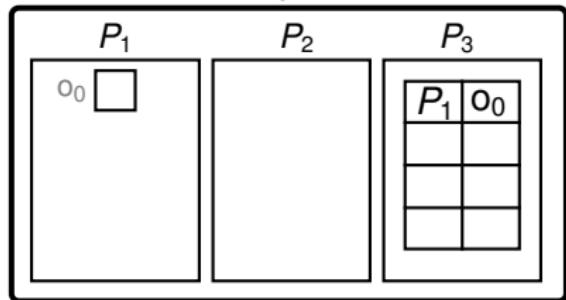


Persistent Medium

Creating a Persistence Point

```
► struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

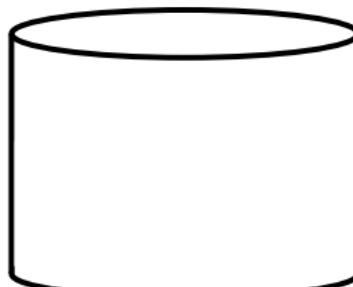
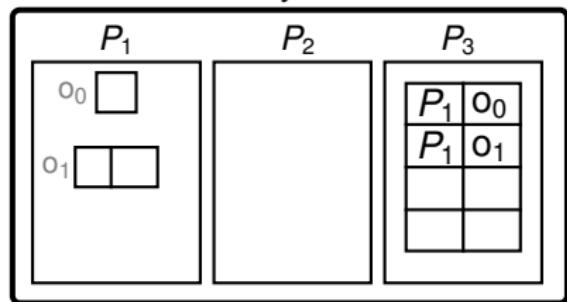


Persistent Medium

Creating a Persistence Point

```
► struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

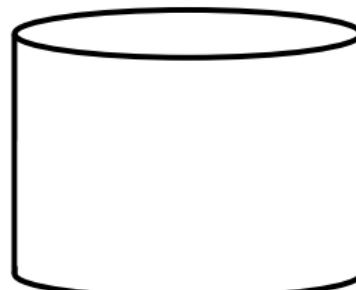
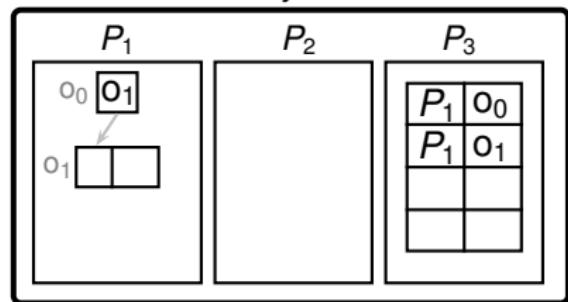


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

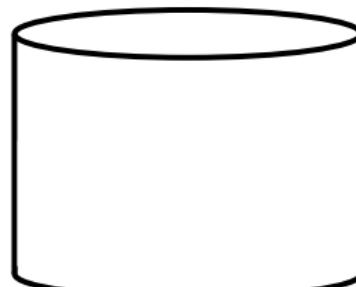
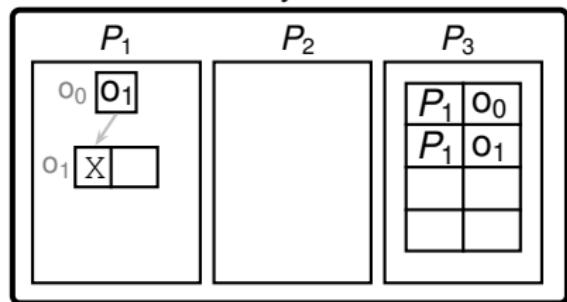


Persistent Medium

Creating a Persistence Point

```
► struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

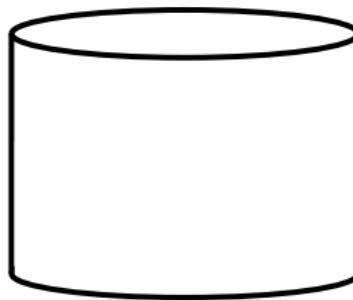
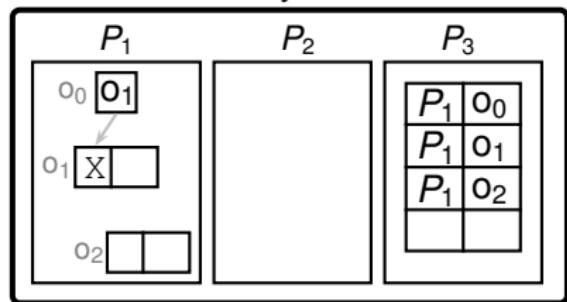


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

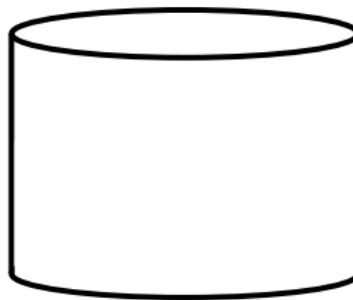
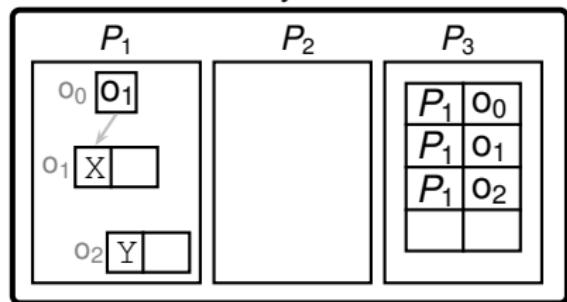


Persistent Medium

Creating a Persistence Point

```
► struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

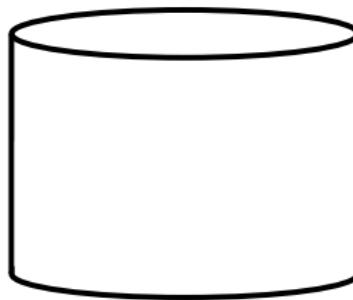
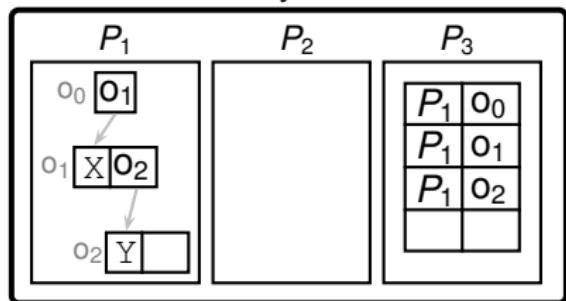


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

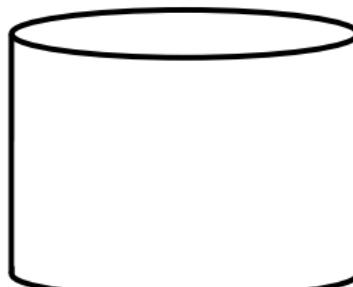
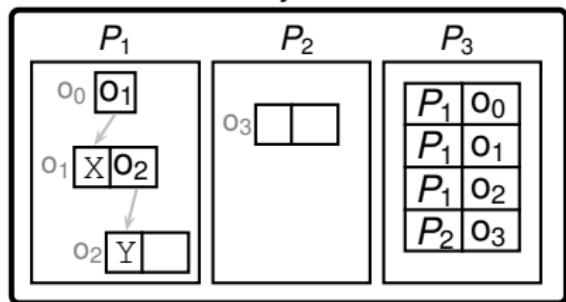


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

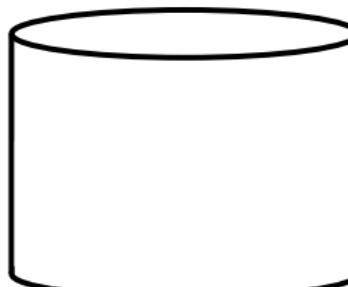
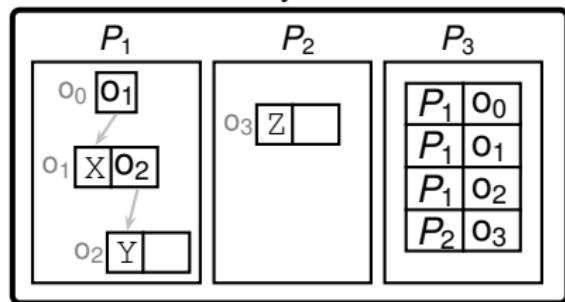


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

Process Memory

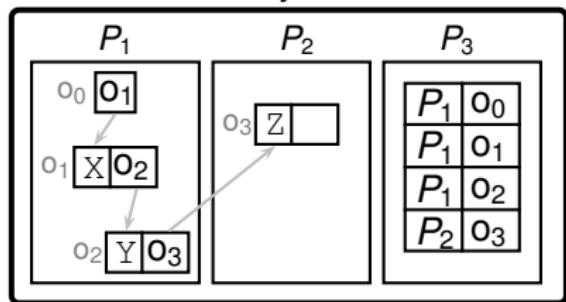


Persistent Medium

Creating a Persistence Point

```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```

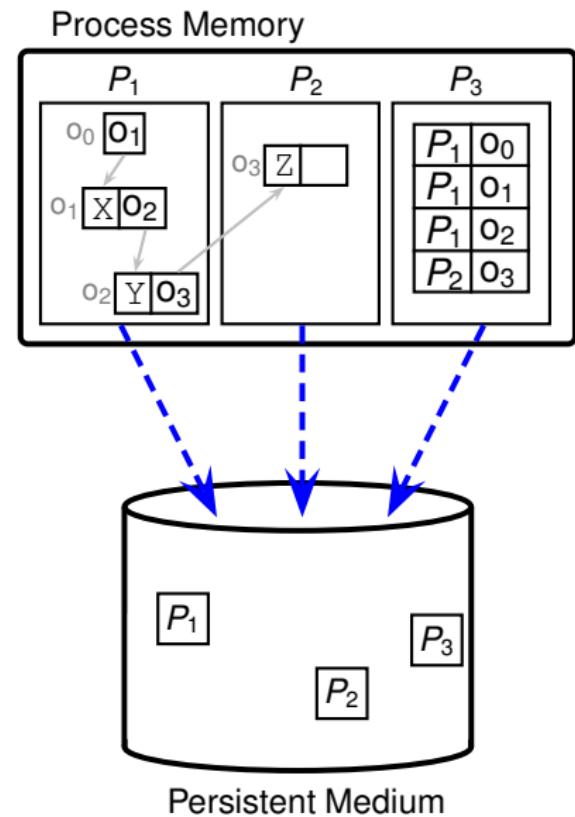
Process Memory



Persistent Medium

Creating a Persistence Point

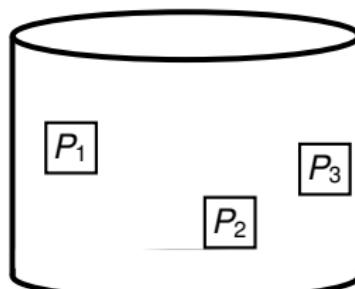
```
struct pcont {  
    list *L;  
} C;  
a() {  
    if (!(cid = pCRestore(&C)))  
        cid = pCAalloc(&C);  
    list *l = malloc(...);  
    C->L = l;  
    l->v = X;  
    list *e1 = malloc(...);  
    e1->v = Y;  
    l->n = e1;  
    list *e2 = malloc(...);  
    e2->v = Z;  
    e1->n = e2;  
    pPoint(cid);  
}  
:  
}
```



Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
► a() {  
    if (!cid = pCRestore(&C))  
        ;  
    ;  
}  
    ;  
}
```

Process Memory



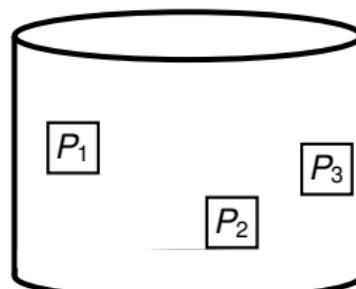
Restoring a Persistence Point



```
struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ;  
    ;  
}  
}
```

Prev	Curr

Process Memory



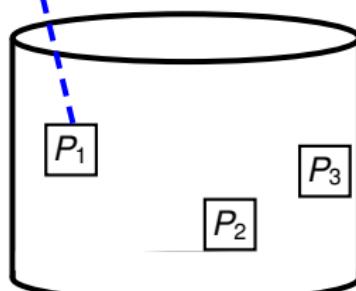
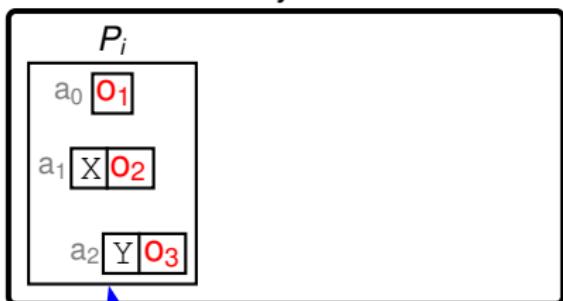
Persistent Medium

Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

Prev	Curr
P_1	P_i

Process Memory



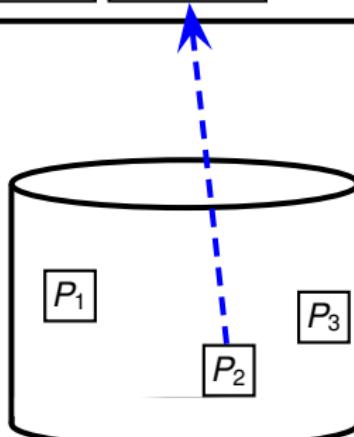
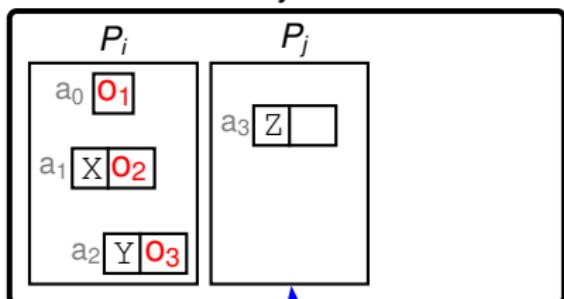
Persistent Medium

Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

Prev	Curr
P_1	P_i
P_2	P_j

Process Memory

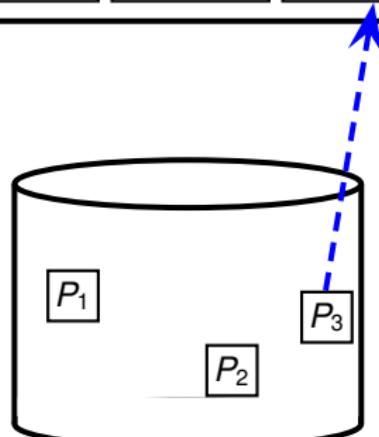
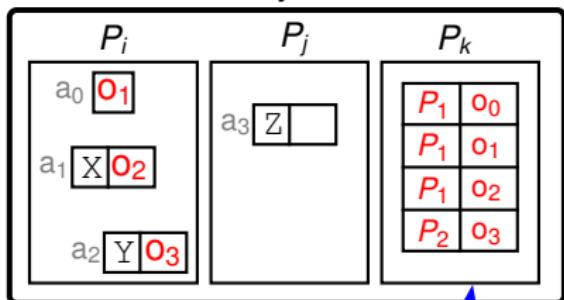


Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory



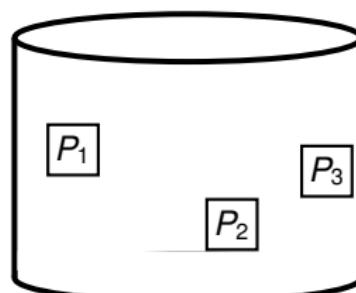
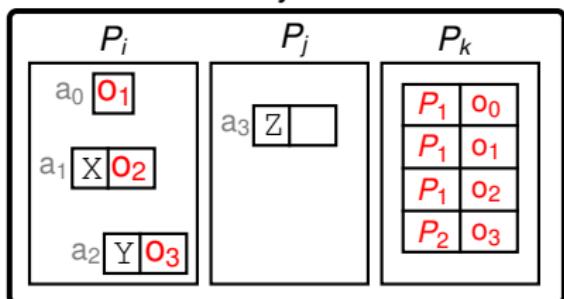
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory



Persistent Medium

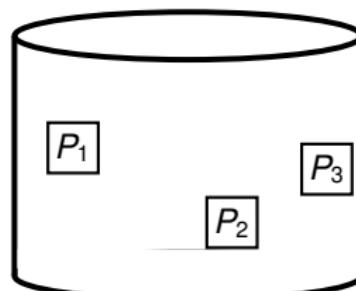
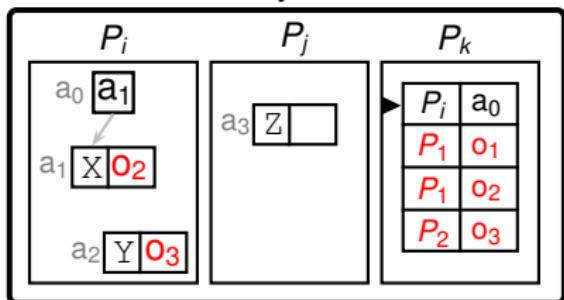
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory



Persistent Medium

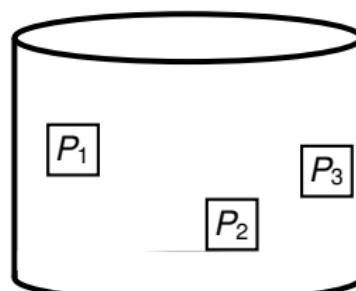
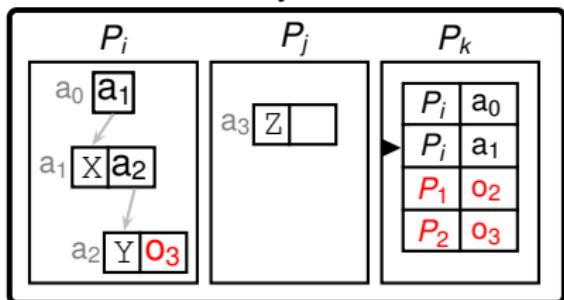
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory



Persistent Medium

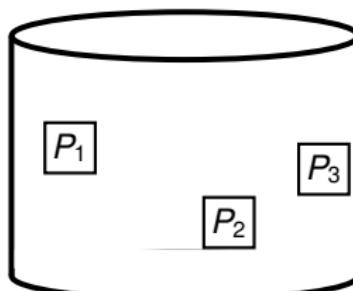
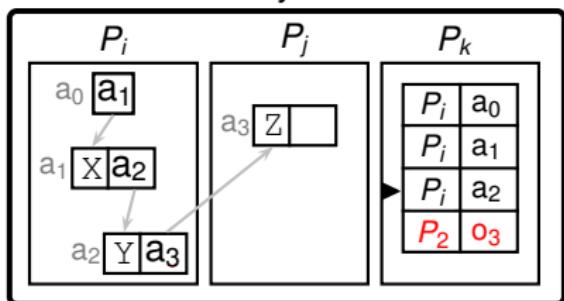
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory

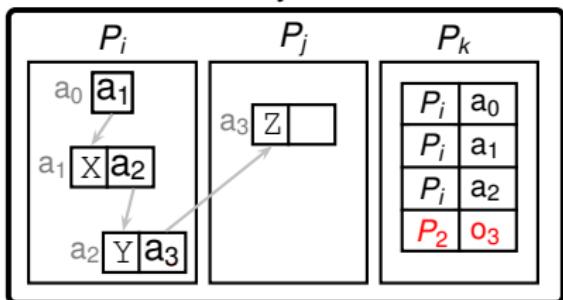


Persistent Medium

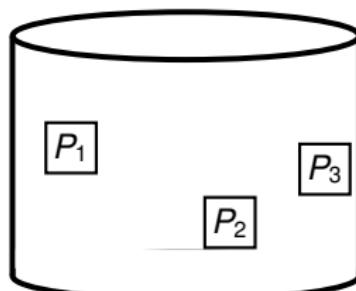
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

Process Memory



Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k



Persistent Medium

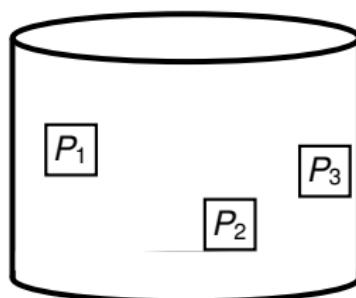
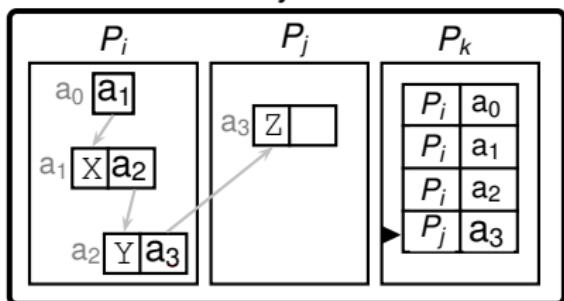
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory



Persistent Medium

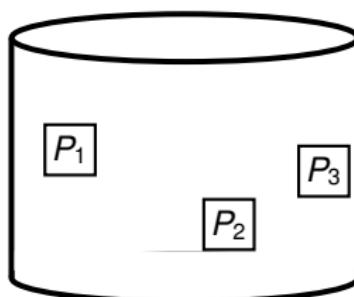
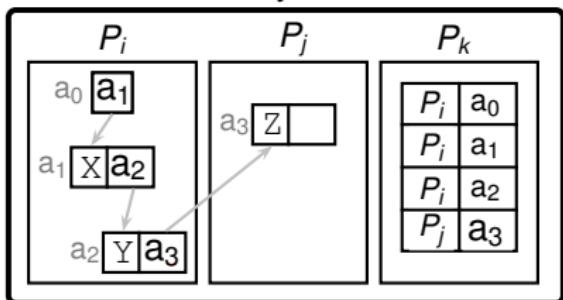
Restoring a Persistence Point

```
► struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

►

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

Process Memory

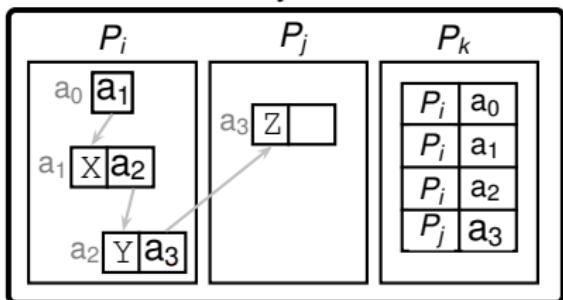


Persistent Medium

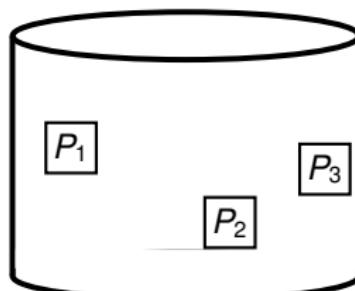
Restoring a Persistence Point

```
struct pcont {  
    list *l;  
} C;  
a() {  
    if (!cid = pCRestore(&C))  
        ...  
    }  
    ...  
}
```

Process Memory



Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k



Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ **Virtualizing container storage (chunk maps)**
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ **Handling out-of-core containers**

Challenges and Solutions

- ▶ Type agnosticism
- ▶ Discovering containers
- ▶ Accurate pointer detection (static and dynamic analysis)
- ▶ Separating the persistent from the volatile
- ▶ Handling memory moves, reallocs, etc.
- ▶ Virtualizing container storage (chunk maps)
- ▶ Version maps
- ▶ Handling out-of-core containers

▶ **Version 1 took us about 18 months.**

Some Results

Configurations

- ▶ **ALP:** Application-level persistence using the file system
- ▶ **SoftPM:** Using SSD or disk drive

Development complexity evaluation

- ▶ Evaluate using basic data structure types
- ▶ Complexity measure using lines of code (LoC)

Performance evaluation

- ▶ Microbenchmarks create and use 1 or more containers
- ▶ MPI Matrix-multiplication and SQLite

Development Complexity

Data Structure	Original LOC	File-based Persistence	Using SoftPM
Hash Table	229		
Linked List	60		
Binary Tree	70		
Parallel MatMult	149		
SQLite	73042		

Table: Lines of code required to make structures persistent and recover them from disk. File-based persistence does not implement atomic transactions for persistence at all. MatMult (*SoftPM* version) uses most LOC to ensure processes restored to same version after crash.

Development Complexity

Data Structure	Original LOC	File-based Persistence	Using SoftPM
Hash Table	229	105	
Linked List	60	20	
Binary Tree	70	54	
Parallel MatMult	149	64	
SQLite	73042	6696	

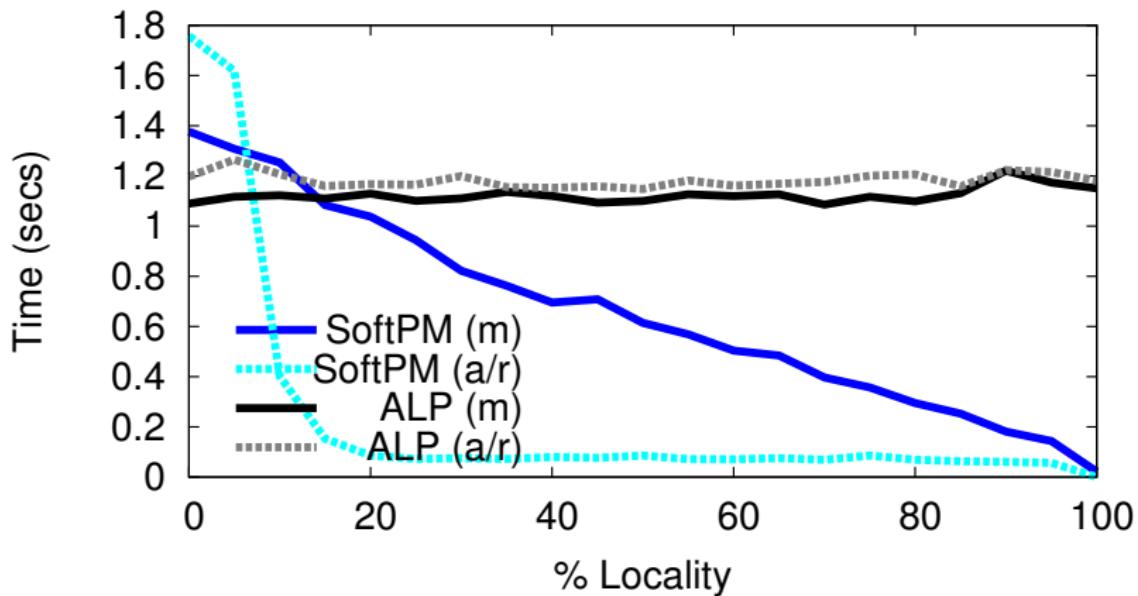
Table: Lines of code required to make structures persistent and recover them from disk. File-based persistence does not implement atomic transactions for persistence at all. MatMult (*SoftPM* version) uses most LOC to ensure processes restored to same version after crash.

Development Complexity

Data Structure	Original LOC	File-based Persistence	Using SoftPM
Hash Table	229	105	4
Linked List	60	20	4
Binary Tree	70	54	4
Parallel MatMult	149	64	33
SQLite	73042	6696	9

Table: Lines of code required to make structures persistent and recover them from disk. File-based persistence does not implement atomic transactions for persistence at all. MatMult (*SoftPM* version) uses most LOC to ensure processes restored to same version after crash.

Incremental checkpointing



4 Candidates
○○○○

Context
○

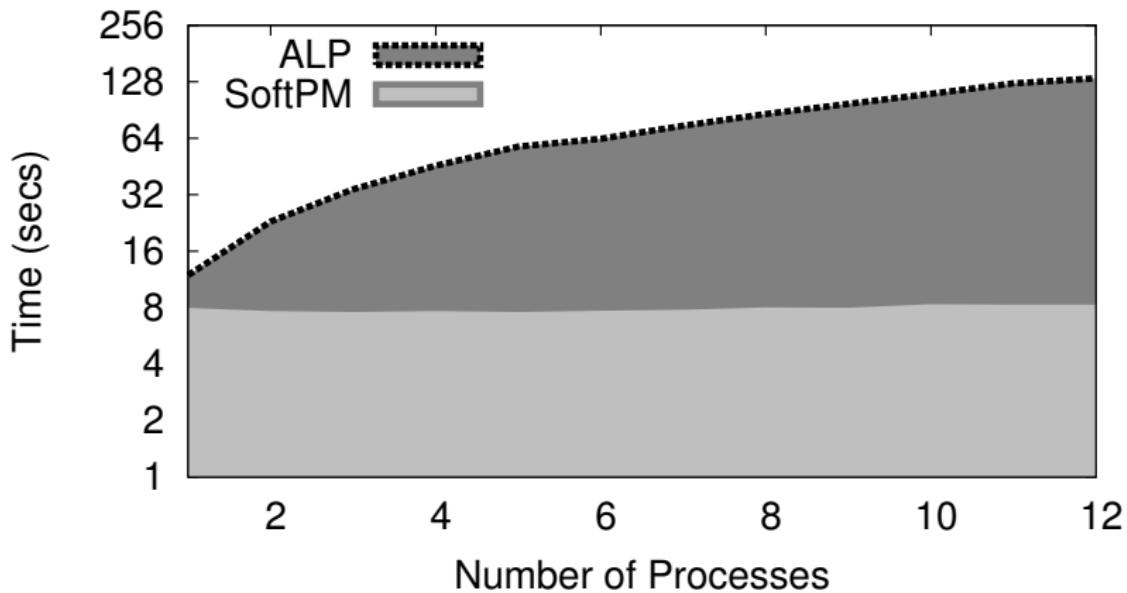
Exploration
○○○

Innovation
○○○○

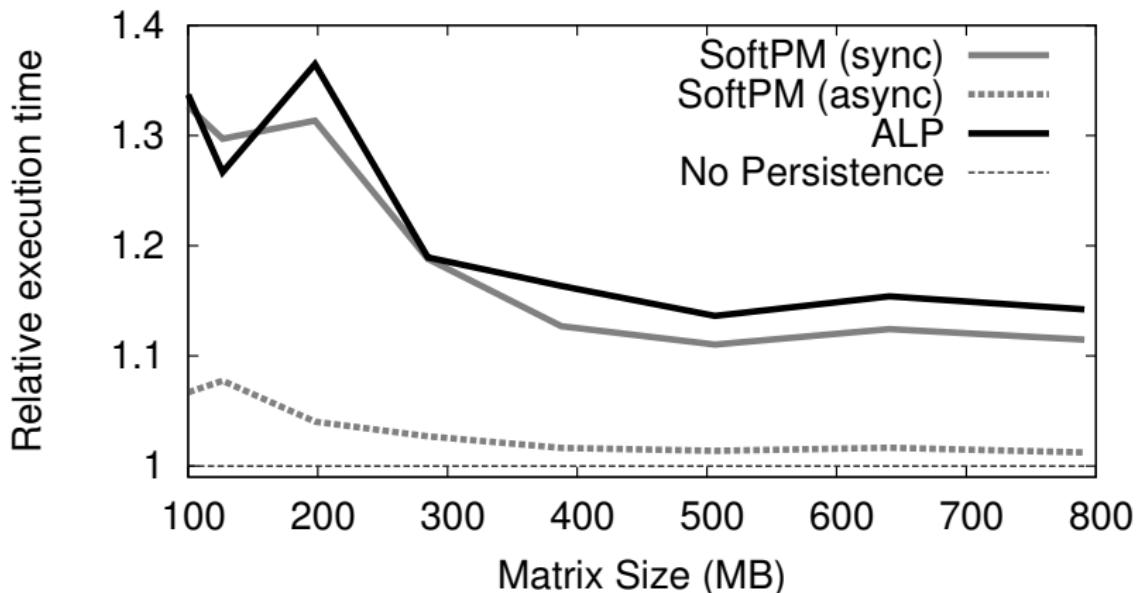
Numbers
○○○○○●○

Summary
○○○

Interleaved Writing



Matrix multiplication with MPI



Summary

A Different Programming Model

- ▶ A memory abstraction for persistent data
- ▶ Fully automated persistence
- ▶ Transactional persistence
- ▶ High-performance persistence I/O
- ▶ Versioning, branching, and out-of-core support

The Team

WEB: <http://sylab.cs.fiu.edu/>

Daniel C.



Daniel G.



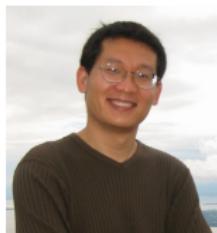
Jorge



Leonardo



Jason



Jinpeng



Ming



Raju



4 Candidates
○○○○

Context
○

Exploration
○○○

Innovation
○○○○

Numbers
○○○○○○○

Summary
○○●

Thank you!

Supported by NSF CCF-0937964

WEB: <http://sylab.cs.fiu.edu/>

raju@cs.fiu.edu